B.I.A.G.R.A.

BIbliotecA de proGRamación científicA

José Angel de Bustos Pérez
<jadebustos@gmail.com>

Version 1.0, December 11, 2016.
LaTeX 2_{ε}

Contents

I Introduction **7**

1 What is *B.I.A.G.R.A* ? **9**
- 1.1 C language? 9
- 1.2 Some general ideas about *B.I.A.G.R.A* 10
- 1.3 Códigos devueltos por las funciones 11
- 1.4 Como instalar *B.I.A.G.R.A* en LiNUX 11
  - 1.4.1 Intalación de la biblioteca *B.I.A.G.R.A* estática 11
  - 1.4.2 Intalación de la biblioteca *B.I.A.G.R.A* dinámica ELF 12
  - 1.4.3 Instalación de ambas bibliotecas 12
- 1.5 Como utilizar *B.I.A.G.R.A* en LiNUX 12
  - 1.5.1 Biblioteca estática 12
  - 1.5.2 Biblioteca dinámica ELF 12

II B.I.A.G.R.A Data structures and constants **13**

2 B.I.A.G.R.A constants (const.h) **15**
- 2.1 Introduction 15
- 2.2 Mathematical constants 15
- 2.3 Logical constants 15
- 2.4 Error constants 16

III Memory allocation **17**

3 Memory allocation (resmem.h) **19**
- 3.1 Introduction 19
- 3.2 Vector's memory allocation 19
  - 3.2.1 `intPtMemAllocVec` function 19
  - 3.2.2 `dblPtMemAllocVec` function 19
- 3.3 Matrix memory allocation 20

3.3.1 dblPtMemAllocMat function 20
3.3.2 dblPtMemAllocUpperTrMat function 20
3.3.3 dblPtMemAllocLowerTrMat function 21
3.4 Freeing memory 22
3.4.1 freeMemDblMat function 22

IV Mathematical functions **23**

4 Pseudo random numbers (random.h) **25**
4.1 Introduction 25
4.2 Pseudo random integer numbers 25
4.2.1 intRandom function 25
4.2.2 uintRandom function 26
4.3 Pseudo random floating point numbers 26
4.3.1 dblRandom function 26
4.3.2 udblRandom function 27

5 Complex numbers (complejo.h) **29**
5.1 Introduction 29
5.2 Data structures 29
5.2.1 biaComplex data structure 29
5.2.2 biaPolar data structure 29
5.3 Arithmetical operations using complex numbers 30
5.3.1 addComplex function 30
5.3.2 subtractComplex function 31
5.3.3 multiplyComplex function 31
5.3.4 divideComplex function 32
5.3.5 invSumComplex function 32
5.3.6 invMulComplex function 33
5.4 Complex number operations 33
5.4.1 dblComplexModulus function 33
5.4.2 dblComplexArg function 33
5.4.3 conjugateComplex function 34
5.4.4 complex2Polar function 34
5.4.5 polar2Complex function 34

6 Integer numbers (integers.h) **35**
6.1 Introduction 35
6.2 Sum integers 35
6.2.1 uintSumFirstNIntegers function 35

6.3 Prime numbers . . . 35
6.3.1 `isPrime` function . . . 35
6.3.2 `getFirstNPrimes` function . . . 36

7 Polynomial (polynomials.h) 37
7.1 Introduction . . . 37
7.2 Data structures . . . 37
7.2.1 `biaRealPol` data structure . . . 37
7.3 Polynomial derivatives . . . 38
7.3.1 `derivativePol` function . . . 38
7.4 Arithmetical operations using polynomials . . . 39
7.4.1 `addPol` function . . . 39
7.4.2 `subtractPol` function . . . 40
7.4.3 `multiplyPol` function . . . 40

8 Matrix (matrix.h) 43
8.1 Introduction . . . 43
8.2 Data structures . . . 43
8.2.1 `biaMatrix` data structure . . . 43
8.3 Matrix creation . . . 44
8.3.1 `identityMatrix` function . . . 44
8.3.2 `scalingMatrix` function . . . 45
8.3.3 `nullMatrix` function . . . 45
8.4 Matrix operations . . . 46
8.4.1 `transposeMatrix` function . . . 46
8.5 Matrix checks . . . 46
8.5.1 `isIdentityMatrix` function . . . 46
8.5.2 `isNullMatrix` function . . . 47
8.5.3 `isSymmetricMatrix` function . . . 47

9 Roots approximation (roots.h) 49
9.1 Introduction . . . 49
9.2 Data structures . . . 49
9.2.1 `biaRealRoot` data structure . . . 49
9.3 Function roots approximation . . . 50
9.3.1 `newtonPol` function . . . 50
9.3.2 `newtonMethod` function . . . 51

10 Runge-Kutta methods (rngkutta.h) 53
10.1 Introduction . . . 53
10.2 Data structures . . . 53

10.2.1 `biaButcherArray` data structure 53
10.2.2 `biaDataRK` data structure 54
10.3 Node number calculations 55
10.3.1 `intNodeNumber` function 55
10.4 Explicit Runge-Kutta methods (scalar problems) 56
10.4.1 `ExplicitRungeKutta` function 56
10.4.2 `RungeKuttaClasico` function 57
10.4.3 MetodoHeun 58
10.4.4 MetodoKutta 58
10.4.5 `EulerModificado` function 59
10.4.6 `EulerMejorado` function 60

A Runge-Kutta methods **63**
A.1 What is a Runge-Kutta method? 63
A.1.1 What is a I.V.P.? 63
A.2 Runge-Kutta's method notation 64
A.2.1 General formulation 64
A.2.2 Matricial notation (Butcher's) 64
A.3 Runge-Kutta types 65
A.3.1 Implicit Runge-Kutta 65
A.3.2 Semi-implicit Runge-Kutta 65
A.3.3 Explicit Runge-Kutta 65

Part I

Introduction

Chapter 1

What is *B.I.A.G.R.A* ?

- *B.I.A.G.R.A* stands for **BI***bliotec***A** *de pro***GR***amación científic***A** which means Scientific Programming Library.
- *B.I.A.G.R.A* is enterely coded using **C** language.
- *B.I.A.G.R.A* has been developed and tested under *LINUX*.
- *B.I.A.G.R.A* is distribuded as open source and its author does not take any responibility.
- I wrote *B.I.A.G.R.A* in the 90s to help me with some subjects in my degree.

1.1 C language?

C language instead of **FORTRAN**?

- **C** is modular and structured.
- **C** is a general purpose language programming.
- **C** is a very powerful language and its code is very fast.
- **C** allows dynamic memory allocation.
- **C** code is portable.
- **C** is able to handle graphic modes.

1.2 Some general ideas about *B.I.A.G.R.A*

B.I.A.G.R.A has been developed under **Linux** and some **Linux** knowledge is needed.

B.I.A.G.R.A was developed to solve general problems instead of particular ones. For instance, instead of writing a program to get the inverse of a 4x4 matrix and having to change the source code to get the inverse of a 5x5 matrix *B.I.A.G.R.A* was developed to allow to write programs to get the inverse of any matrix without having to change de sorce code.

To be able to do that *pointers* were used instead of using *arrays*.

When we talk about *vectors* we will be talking about a *pointer* using dynamic memory allocation. When we talk about matrices we will be talking about *pointer* to a *pointer* using dynamic memory allocation.

B.I.A.G.R.A uses some data structures to store data.

For common errors as:

- Errors in dynamic memory allocation.
- Division by zero.
- ...

B.I.A.G.R.A uses its own constants to notify these errors (Chapter 2).

Cuando una función devuelva un dato, que no sea un código que indique el estado en el que se terminó la ejecución de la función, se indicará anteponiendo un prefijo al nombre de la función, el cual indicará que tipo de dato devuelve, por ejemplo:

1. *double dblFuncion(...)* función que devuelve un dato de tipo *double*.
2. *int intFuncion(...)* función que devuelve un dato de tipo *int*.
3. *double *dblPtFuncion(...)* función que devuelve un dato de tipo puntero a *double*.
4. *void Funcion(...)* función que no devuelve ningún dato.

1.3 Códigos devueltos por las funciones

No es obligatorio que las funciones devuelvan un valor.
Cuando una función devuelva un valor será por dos razones:

- Para devolver el resultado de una operación.
- Para informar de como terminó una operación.

Es este último caso el que nos incumbe.

Cuando una función devuelva un dato indicando como terminó una determinada operación, este dato será, necesariamente, un entero y los códigos devueltos los podemos ver en la página **??**.

1.4 Como instalar *B.I.A.G.R.A* en LiNUX

Para instalar *B.I.A.G.R.A* lo primero que hay que hacer es entrar en el sistema como **root** y situarse en el directorio donde esten los fuentes de la biblioteca.

1.4.1 Intalación de la biblioteca *B.I.A.G.R.A* estática

Para instalar este tipo bibilioteca se puede hacer de dos formas:

1. *./instalar estatica*
2. *make estatica*

En realidad ambas hacen lo mismo, la opción con *make* en realidad ejecuta *./instalar estatica*.

Al realizar cualquiera de estas dos opciones se copiarán los ficheros cabezera a ***/usr/include/biagra*** y a continuación se creará la biblioteca ***/usr/lib/libbiagra.a***.

Luego si se utiliza una función de esta biblioteca, cuyo prototipo está en el fichero de cabecera *rngkutta.h* habrá que incluir en las directivas al prepocesador **#include <biagra/rngkutta.h>**.

1.4.2 Intalación de la biblioteca *B.I.A.G.R.A* dinámica ELF

1.4.3 Instalación de ambas bibliotecas

Para instalar la biblioteca *B.I.A.G.R.A* en su forma estática y dinámica ELF:

make todo

Esto lo que hace es ejecutar primero

./instalar estatica

lo cual instalará la biblioteca estática, y luego

./instalar elf

lo cual instalará la biblioteca dinámica ELF.

1.5 Como utilizar *B.I.A.G.R.A* en LiNUX

B.I.A.G.R.A es una biblioteca para programación científica, desarrollada para ser usada en programas escritos en **C**, se distribuye en varios formatos:

Biblioteca estática

Biblioteca dinámica ELF

1.5.1 Biblioteca estática

Para el uso de esta biblioteca hay que indicarle al *montador* que biblioteca debe *enlazar*. Por ejemplo, supongamos que hemos escrito un programa para resolver una ecuación diferencial por un método *Runge-Kutta* y hemos utilizado funciones cuyos prototipos estan en **edo.h** y **rngkutta.h**, si nuestro programa es *programa.c*, para crear el ejecutable:

*gcc programa.c -o programa -l**biagra** -lm*[1]

1.5.2 Biblioteca dinámica ELF

[1] *B.I.A.G.R.A* utiliza la biblioteca estandar matemática.

Part II

B.I.A.G.R.A Data structures and constants

Chapter 2

B.I.A.G.R.A constants (const.h)

2.1 Introduction

B.I.A.G.R.A includes its own constants to be used if needed.

These constants are defined in `const.h`.

2.2 Mathematical constants

Table 2.1 shows the *B.I.A.G.R.A* 's mathematical constanst.

Constant	Name	Value
e	`BIA_E`	2.71828182845904523536029
π	`BIA_PI`	3.14159265358979323846264

Table 2.1: *B.I.A.G.R.A* mathematical constants.

2.3 Logical constants

The following logical constants are defined:

BIA_FALSE when a condition is not met.

BIA_TRUE when a condiction is met.

2.4 Error constants

The following error constants are defined:

BIA_ZERO_DIV division by zero.

BIA_MEM_ALLOC error in memory allocation.

Part III

Memory allocation

Chapter 3

Memory allocation (resmem.h)

3.1 Introduction

B.I.A.G.R.A includes its own memory allocation functions which are defined in `resmem.h` file.

3.2 Vector's memory allocation

Some functions are provided to handle memory allocations for vectors.

3.2.1 `intPtMemAllocVec` function

This functions allocates memory for a vector of int.

The definition of this function:

```
double *intPtMemAllocVec(int intElements);
```

This function has only one argument, `intElements`, which is the dimension of the vector and a `int` pointer is returned.

3.2.2 `dblPtMemAllocVec` function

This functions allocates memory for a vector of doubles.

The definition of this function:

```
double *dblPtMemAllocVec(int intElements);
```

This function has only one argument, `intElements`, which is the dimension of the vector and a `double` pointer is returned.

Draft

3.3 Matrix memory allocation

Some functions are provided to handle memory allocations for vectors.

3.3.1 `dblPtMemAllocMat` function

This function allocates memory for a matrix of doubles.

The definition of this function:

```
double **dblPtMemAllocMat(int intRows, int intCols);
```

where:

intRows number of rows.

intCols number of columns.

3.3.2 `dblPtMemAllocUpperTrMat` function

This function allocates memory for a upper triangular square matrix.

The definition of this function:

```
double **dblPtMemAllocUpperTrMat(int intOrder);
```

This function has only one argument, `intOrder`, which is the order of the matrix and a `double` pointer to pointer is returned.

In a upper triangular square matrix all elements below the diagonal are zero:

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} \\ 0 & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ 0 & 0 & a_{2,2} & a_{2,3} & a_{2,4} \\ 0 & 0 & 0 & a_{3,3} & a_{3,4} \\ 0 & 0 & 0 & 0 & a_{4,4} \end{pmatrix}$$

For `intOrder = 5`:

```
myMatrix = dbpPtMemAllocUpperTrMat(5);
```

and:

Pointer	# elements	First element	Last element
`myMatrix[0]`	5	0	4
`myMatrix[1]`	4	0	3
`myMatrix[2]`	3	0	2
`myMatrix[3]`	2	0	1
`myMatrix[4]`	1	0	0

so:

$$myMatrix[i][j] = *(*(myMatrix + i) + j) = \begin{cases} a_{i,j+i} & \forall\, i \leq j \\ 0 & \forall\, i > j \end{cases}$$

3.3.3 `dblPtMemAllocLowerTrMat` function

This function allowcates memory for a lower triangular square matrix.

The definition of this function:

```
double **dblPtMemAllocLowerTrMat(int intOrder);
```

This function has only one argument, `intOrder`, which is the order of the matrix and a `double` pointer to pointer is returned.

In a lower triangular square matrix all elements above the diagonal are zero:

$$\begin{pmatrix} a_{0,0} & 0 & 0 & 0 & 0 \\ a_{1,0} & a_{1,1} & 0 & 0 & 0 \\ a_{2,0} & a_{2,1} & a_{2,2} & 0 & 0 \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & 0 \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix}$$

For `intOrder = 5`:

```
myMatrix = dbpPtMemAllocLowerTrMat(5);
```

and:

Pointer	# elements	First element	Last element
`myMatrix[0]`	1	0	0
`myMatrix[1]`	2	0	1
`myMatrix[2]`	3	0	2
`myMatrix[3]`	4	0	3
`myMatrix[4]`	5	0	4

so:

$$myMatrix[i][j] = *(*(myMatrix + i) + j) = \begin{cases} a_{i,j} & \forall\ i \leq j \\ 0 & \forall\ i < j \end{cases}$$

3.4 Freeing memory

B.I.A.G.R.A includes its own functions to free memory.

3.4.1 `freeMemDblMat` function

Part IV

Mathematical functions

Chapter 4

Pseudo random numbers (random.h)

4.1 Introduction

B.I.A.G.R.A includes its own functions to pseudo random number generation and they are defined in `random.h` file.

This functions have not been tested to produce unpredictable sequences, so be careful when use them.

4.2 Pseudo random integer numbers

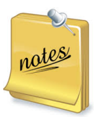
4.2.1 `intRandom` function

This function generates random integers.

The definition of this function:

```
int intRandom(int limit);
```

The pseudo random integer is placed in the interval $(-limit, limit)$.



Before using this function ***`srand`*** *must be used to initialize* ***`rand`****. You can use* ***`srand((unsigned)time(NULL))`****.*

The pseudo random number is generated with the following formula:

$$\left[\frac{limit \cdot rand()}{RAND\_MAX + 1}\right] \in (-limit, limit)$$

Then randomly is choosed if the number is positive or negative using the above formula with $limit = 2$ and then taking modulus 2. If modulus is 1 then the number will be a negative one.

4.2.2 `uintRandom` function

This function generates random integers.

The definition of this function:

```
int uintRandom(int limit);
```

The pseudo random integer is placed in the interval $[0, limit)$.

Before using this function ***`srand`*** *must be used to initialize* ***`rand`****. You can use* ***`srand((unsigned)time(NULL))`****.*

The pseudo random number is generated with the following formula:

$$\left[\frac{limit \cdot rand()}{RAND\_MAX + 1}\right] \in [0, limit)$$

4.3 Pseudo random floating point numbers

4.3.1 `dblRandom` function

This function generates random floating point numbers.

The definition of this function:

```
int dblRandom(int limit);
```

The pseudo random floating point number is placed in the interval $(-limit, limit)$.

Before using this function ***`srand`*** *must be used to initialize* ***`rand`****. You can use* ***`srand((unsigned)time(NULL))`****.*

The pseudo random number is generated with the following formula:

$$\frac{limit \cdot rand()}{RAND\_MAX + 1} \in (-limit, limit)$$

Then randomly is choosed if the number is positive or negative using the above formula with $limit = 2$ and then taking modulus 2. If modulus is 1 then the number will be a negative one.

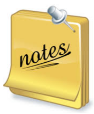
4.3.2 `udblRandom` function

This function generates random floating point numbers.

The definition of this function:

```
int udblRandom(int limit);
```

The pseudo random floating point number is placed in the interval $[0, limit)$.



Before using this function ***`srand`*** *must be used to initialize* ***`rand`****. You can use* ***`srand((unsigned)time(NULL))`****.*

The pseudo random number is generated with the following formula:

$$\frac{limit \cdot rand()}{RAND\_MAX + 1} \in [0, limit)$$

Chapter 5

Complex numbers (complejo.h)

5.1 Introduction

Functions to manage complex numbers are defined in `complex.h` file.

5.2 Data structures

Some data structures are defined in *B.I.A.G.R.A* to manage complex numbers.

5.2.1 `biaComplex` data structure

This data structure is used to handle polinomials $p(x) \in \mathbb{R}[x]$. **biaComplex** data structure is defined in figure 7.1 where:

intDegree polynomial degree.

intRealRoots number of real roots (if any).

intCompRoots number of complex roots (if any).

***dblCoef** pointer to store polynomial coeficients.

5.2.2 `biaPolar` data structure

This data structure is used to store data for root approximation. Data structure is defined in figure 9.1 where:

```
typedef struct {
  double dblReal,
         dblImag;
  } biaComplex;
```

Figure 5.1: biaComplex data structure.

intNMI maximum number of iterations to get the root with a maximum error of *dblTol*.

intIte iterations used to get the root.

dblx0 initial approximation to get the root.

dblRoot root approximation.

dblTol maximum tolerance when calculating the root.

dblError error in root approximation. Difference between the las two root approximations.

```
typedef struct {
  double dblMod,
         dblArg;
  } biaPolar;
```

Figure 5.2: biaPolar data structure.

5.3 Arithmetical operations using complex numbers

5.3.1 `addComplex` function

This function adds two complex numbers.

The definition of this function:

```
void addComplex(biaComplex *ptCmplx1, biaComplex *ptCmplx2,
                biaComplex *ptRes);
```

where:

***ptCmplx1** first complex number to be added.

***ptCmplx2** second complex number to be added.

***ptRes** result of the operation.

5.3.2 `subtractComplex` function

This function subtracts two complex numbers.

The definition of this function:

```
void subtractComplex(biaComplex *ptCmplx1, biaComplex *ptCmplx2,
                     biaComplex *ptRes);
```

where:

***ptCmplx1** complex number.

***ptCmplx2** complex number to be subtracted to the above.

***ptRes** result of the operation.

5.3.3 `multiplyComplex` function

This function multiplies two complex numbers.

The definition of this function:

```
void multiplyComplex(biaComplex *ptCmplx1, biaComplex *ptCmplx2,
                     biaComplex *ptRes);
```

where:

ptCmplx1 first complex number to be multiplied.

ptCmplx2 second complex number to be multiplied.

ptRes result of the operation.

5.3.4 `divideComplex` function

This function divides one complex number by other:

$$\frac{\mathrm{a} + \mathrm{b} \cdot i}{\mathrm{c} + \mathrm{d} \cdot i} = (\mathrm{a} + \mathrm{b} \cdot i) \cdot (\mathrm{c} + \mathrm{d} \cdot i)^{-1}$$

The definition of this function:

```
int divideComplex(biaComplex *ptCmplx1, biaComplex *ptCmplx2,
                  biaComplex *ptRes);
```

where:

***ptCmplx1** complex number.

***ptCmplx2** complex number used as divisor.

***ptRes** result of the operation.

The following codes are returned:

BIA_ZERO_DIV	Division by zero
BIA_TRUE	Success

5.3.5 `invSumComplex` function

This function gets the additive inverse of a complex number:

$$\forall \quad z_1 \in \mathbb{C} \quad \exists \quad z_2 \in \mathbb{C} \quad | \quad z_1 + z_2 = 0$$

The definition of this function:

```
void invSumComplex(biaComplex *ptCmplx, biaComplex *ptRes);
```

where:

***ptCmplx** complex number to get its additive inverse.

***ptRes** where the additive inverse will be stored.

5.3.6 `invMulComplex` function

This function gets the multiplicative inverse of a complex number:

$$\forall \;\; z_1 \in \mathbb{C} - \{0\} = \mathbb{C}^* \quad \exists \;\; z_2 \in \mathbb{C} \;\; | \;\; z_1 \cdot z_2 = 1$$

The definition of this function:

```
int invMulComplex(biaComplex *ptCmplx, biaComplex *ptRes) ;
```

where:

***ptCmplx** complex number to get its multiplicative inverse.

***ptRes** where the additive multiplicative will be stored.

The following codes are returned:

BIA_ZERO_DIV	Division by zero
BIA_TRUE	Success

5.4 Complex number operations

5.4.1 `dblComplexModulus` function

This function gets the modulus of a complex number.

The definition of this function:

```
double dblComplexModule(biaComplex *ptCmplx);
```

where:

***ptCmplx** complex number to get its modulus.

This function returns the complex number modulus.

5.4.2 `dblComplexArg` function

This function gets the argument of a complex number.

The definition of this function:

```
double dblComplexArg(biaComplex *ptCmplx);
```

where:

***ptCmplx** complex number to get its argument.

This function returns the complex number argument (radians).

5.4.3 `conjugateComplex` function

This function gets the conjugate complex of a complex number:

$$z = \mathrm{a} + \mathrm{b} \cdot i \quad \in \mathbb{C} \Rightarrow \overline{z} = \mathrm{a} - \mathrm{b} \cdot i \quad \in \mathbb{C}$$

The definition of this function:

```
void conjugateComplex(biaComplex *ptCmplx, biaComplex *ptRes);
```

where:

***ptCmplx** complex number to get its conjugate.

***ptRes** complex conjugate.

5.4.4 `complex2Polar` function

This function gets the polar coordinates of a complex number.

The definition of this function:

```
void complex2Polar(biaComplex *ptCmplx, biaPolar *ptRes);
```

where:

***ptCmplx** complex number to calculate polar coordinates.

***ptRes** polar coordinates.

5.4.5 `polar2Complex` function

This function gets the cartesian coordinates of a polar coordinates for a complex number.

The definition of this function:

```
void polar2Complex(biaPolar *ptPolar, biaComplex *ptRes);
```

where:

***ptPolar** polar coordinates.

***ptRes** complex number in cartesian coordinates.

Argument is supposed to be in radians.

Chapter 6

Integer numbers (integers.h)

6.1 Introduction

B.I.A.G.R.A includes functions about integer numbers in `integers.h` file.

6.2 Sum integers

6.2.1 `uintSumFirstNIntegers` function

This function gets the sum of the first n integers.

The definition of this function:

```
unsigned uintSumFirstNIntegers(int n);
```

If the sum is bigger than an unsigned int 0 is returned.

6.3 Prime numbers

6.3.1 `isPrime` function

This function checks if a number is a prime number.

The definition of this function:

```
int isPrime(int intN);
```

The following codes are returned:

BIA_FALSE	`intN` is not a prime number
BIA_TRUE	`intN` is a prime number

6.3.2 `getFirstNPrimes` function

This function checks if a number is a prime number.

The definition of this function:

```
void getFirstNPrimes(unsigned int *ptPrimes, int intNumber, int *ptCalc);
```

where:

***ptPrimes** array where primes will be stored. Memory allocation for this array has to be initialized before using this function.

intNumber number of primes to be computed.

***ptCalc** in this variable the total amount of computed primes will be stored.

Chapter 7

Polynomial (polynomials.h)

7.1 Introduction

Functions to manage polynomials are defined in `polynomial.h` file.

A polynomial used to be represented as shown in equation 7.1.

$$p(x) = a_0 + a_1 \cdot x + \cdots + a_n \cdot x^n = \sum_{i=0}^{n} a_i \cdot x^i \quad \text{where } a_i \in \mathbb{R} \tag{7.1}$$

7.2 Data structures

Some data structures are defined in *B.I.A.G.R.A* to manage polynomials.

7.2.1 `biaRealPol` data structure

This data structure is used to handle polinomials $p(x) \in \mathbb{R}[x]$. **biaPol** data structure is defined in figure 7.1 where:

intDegree polynomial degree.

intRealRoots number of real roots (if any).

intCompRoots number of complex roots (if any).

***dblCoef** pointer to store polynomial coeficients.

```
typedef struct {
  int  intDegree     = 0,
       intRealRoots = 0,
       intCompRoots = 0;

  double  *dblCoefs;
  } biaRealPol;
```

Figure 7.1: biaRealPol data structure.

Polynomial coeficients are stored in `dblCoefs` pointer which has to be previously initialized:

$$
\begin{array}{rcl}
\mathrm{dblCoefs}[0] & = & a_0 \\
\mathrm{dblCoefs}[1] & = & a_1 \\
\ldots & \ldots & \ldots \\
\mathrm{dblCoefs}[n] & = & a_n
\end{array}
$$

7.3 Polynomial derivatives

7.3.1 `derivativePol` function

This function gets the n-th derivative of a polynomial.

The definition of this function:

```
int derivativePol(biaPol *ptPol, biaPol *ptDer, int intN);
```

where:

***ptPol** pointer to a `biaRealPol` struct with the polynomial to get its derivative is stored.

***ptDer** pointer to a `biaRealPol` struct where the derivative will be stored.

intN order of the derivative to get.

The following codes are returned:

BIA_MEM_ALLOC	Memory allocation error
BIA_TRUE	Success

`ptDer` will be released and memory allocation will be carried out to store the derivative.

*`ptDer` member `dblCoefs` has to be initialized to a NULL pointer to avoid a **Segment Fault** error if it was not previously initialized.*

7.4 Arithmetical operations using polynomials

7.4.1 `addPol` function

This function adds two polynomials.

The definition of this function:

```
int addPol(biaPol *ptPol1, biaPol *ptPol2, biaPol *ptRes);
```

where:

***ptPol1** pointer to a `biaPol` struct with the first polynomial to be added.

***ptPol2** pointer to a `biaPol` struct with the second polynomial to be added.

***ptRes** pointer to a `biaPol` struct where the add operation will be stored.

The following codes are returned:

BIA_MEM_ALLOC	Memory allocation error
BIA_TRUE	Success

`ptRes` will be released and memory allocation will be carried out to store the derivative.

*`ptRes` member `dblCoefs` has to be initialized to a NULL pointer to avoid a **Segment Fault** error if it was not previously initialized.*

7.4.2 `subtractPol` function

This function subtracts two polynomials.

The definition of this function:

```
int subtractPol(biaPol *ptPol1, biaPol *ptPol2, biaPol *ptRes);
```

where:

***ptPol1** pointer to a `biaPol` struct with the first polynomial.

***ptPol2** pointer to a `biaPol` struct with the polynomial to be subtracted from the above.

***ptRes** pointer to a `biaPol` struct where the subtract operation will be stored.

The following codes are returned:

BIA_MEM_ALLOC	Memory allocation error
BIA_TRUE	Success

`ptRes` will be released and memory allocation will be carried out to store the derivative.

*`ptRes` member `dblCoefs` has to be initialized to a NULL pointer to avoid a **Segment Fault** error if it was not previously initialized.*

7.4.3 `multiplyPol` function

This functions multiplies two polynomials.

The definition of this function:

```
int subtractPol(biaPol *ptPol1, biaPol *ptPol2, biaPol *ptRes);
```

where:

***ptPol1** pointer to a `biaPol` struct with the first polynomial.

***ptPol2** pointer to a `biaPol` struct with the second polynomial.

***ptRes** pointer to a `biaPol` struct where the multiplication operation will be stored.

The following codes are returned:

BIA_MEM_ALLOC	Memory allocation error
BIA_TRUE	Success

`ptRes` will be released and memory allocation will be carried out to store the derivative.

*`ptRes` member `dblCoefs` has to be initialized to a NULL pointer to avoid a **Segment Fault** error if it was not previously initialized.*

Chapter 8

Matrix (matrix.h)

8.1 Introduction

Functions to manage matrices are defined in `matrix.h` file.

8.2 Data structures

Some data structures are defined in *B.I.A.G.R.A* to manage matrices.

8.2.1 `biaMatrix` data structure

This data structure is used to store a matrix. **biaMatrix** data structure is defined in figure 8.1 where:

intRows number of rows.

intCols number of columns.

****dblCoefs** pointer to store matrix coeficients.

```
typedef struct {
  int intRows,
      intCols;

  double **dblCoefs;
  } biaMatrix;
```

Figure 8.1: biaMatrix data structure.

8.3 Matrix creation

B.I.A.G.R.A includes functions to create some kind of matrices.

8.3.1 `identityMatrix` function

This function stores the identity matrix with order taken from intRows member of `ptMatrix`:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & \ddots & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \ddots & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The definition of this function:

```
void identityMatrix(biaMatrix *ptMatrix);
```

where:

***ptMatrix** matrix that has to be created before using this function. Memory allocation for `dblCoefs` must be done before using this function.

`intRows` is used to get the matrix order.

8.3.2 `scalingMatrix` function

This function stores the scaling matrix with factor λ and order taken from intRows member of `ptMatrix`:

$$\begin{pmatrix} \lambda & 0 & 0 & 0 & 0 \\ 0 & \lambda & \ddots & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \ddots & \lambda & 0 \\ 0 & 0 & 0 & 0 & \lambda \end{pmatrix}$$

The definition of this function:

```
void scalingMatrix(biaMatrix *ptMatrix, double dblFactor);
```

where:

***ptMatrix** matrix that has to be created before using this function. Memory allocation for `dblCoefs` must be done before using this function.

`intRows` is used to get the matrix order.

8.3.3 `nullMatrix` function

This function stores the null matrix with order taken from intRows member of `ptMatrix`:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The definition of this function:

```
void nullMatrix(biaMatrix *ptMatrix);
```

where:

***ptMatrix** matrix that has to be created before using this function. Memory allocation for `dblCoefs` must be done before using this function.

`intRows` and `intCols` is used to get the matrix order.

8.4 Matrix operations

8.4.1 `transposeMatrix` function

This function stores the transpose matrix of a given matrix.

The definition of this function:

```
void transposeMatrix(biaMatrix *ptMatrix, biaMatrix *ptRes);
```

where:

***ptMatrix** matrix to get its transpose matrix.

***ptRes** matrix to store the transpose matrix. Memory has to be preallocated before using this function.

`intRows` and `intCols` is used to get the matrix order.

8.5 Matrix checks

8.5.1 `isIdentityMatrix` function

This function checks if a matrix is the identity matrix.

The definition of this function:

```
int isIdentityMatrix(biaMatrix *ptMatrix);
```

where:

***ptMatrix** matrix to check.

8.5.2 `isNullMatrix` function

This function checks if a matrix is a null matrix.

The definition of this function:

```
int isNullMatrix(biaMatrix *ptMatrix, double dblTol);
```

where:

***ptMatrix** matrix to check.

dblTol if a matrix element is minor than this value it is assumed it is a null element.

8.5.3 `isSymmetricMatrix` function

This function checks if a matrix is a symmetric matrix.

The definition of this function:

```
int isSymmetricMatrix(biaMatrix *ptMatrix);
```

where:

***ptMatrix** matrix to check.

Chapter 9

Roots approximation (roots.h)

9.1 Introduction

Functions to compute function's roots approximation are defined in `roots.h` file.

9.2 Data structures

Some data structures are defined in *B.I.A.G.R.A* to manage roots.

9.2.1 `biaRealRoot` data structure

This data structure is used to store data for root approximation.

Data structure is defined in figure 9.1 where:

intNMI maximum number of iterations to get the root with a maximum error of `dblTol`.

intIte iterations used to get the root.

dblx0 initial approximation to get the root.

dblRoot root approximation.

dblTol maximum tolerance when calculating the root.

dblError error in root approximation. Difference between the las two root approximations.

```
typedef struct {
  int intNMI,
      intIte;

  double dblx0,
         dblRoot,
         dblTol,
         dblError;
  } biaRealRoot;
```

Figure 9.1: biaRealRoot data structure.

9.3 Function roots approximation

9.3.1 `newtonPol` function

This function approaches a polynomial root using the **Newton** method.

The definition of this function:

```
int newtonPol(biaPol *ptPol, biaRealRoot *ptRoot);
```

The following codes are returned:

BIA_MEM_ALLOC	Memory allocation error
BIA_ZERO_DIV	Division by zero
BIA_TRUE	the root was computed satisfying the problem conditions (`intMNI` and `dblTol`);
BIA_FALSE	Root approximation could not be calculated satisfying the requirements (`intMNI` and `dblTol`).

The following values in `*ptRoot` need to be initialized:

intMNI maximun number of iterations to compute the root.

dblx0 initial approximation.

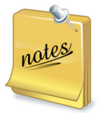
dblTol tolerance to compute de root.

The following data will be stored:

intIte iterations used to compute the root.

dblRoot approximation of the root.

dblError error in the approximation.



When two consecutive approximations are close enough, `dblTol`, last approximation will be considered as good and will be stored in `*biaRealRoot *ptRoot` in `dblRoot`.

9.3.2 `newtonMethod` function

This function approaches a function's root using the **Newton** method.

The definition of this function:

```
int newtonMethod(biaRealRoot *ptRoot,
       int (*func)(double dblx0, double *ptRes),
       int (*der)(double dblx0, double *ptRes));
```

Function's pointers are used to avoid having to recode the C function every time a root function need to be approximated for different mathematical functions.

The following codes are returned:

BIA_ZERO_DIV	Division by zero
BIA_TRUE	the root was computed satisfying the problem conditions (`intMNI` and `dblTol`);
BIA_FALSE	Root approximation could not be calculated satisfying the requirements (`intMNI` and `dblTol`).

where:

***ptRoot** is a pointer to a `biaRealRoot` variable.

***func** pointer to a function implementing the function which root is going to be computed.

***der** pointer to a function implementing the derivative of the function which root is going to be computed.

The following values in `*ptRoot` need to be initialized:

intMNI maximun number of iterations to compute the root.

dblx0 initial approximation.

dblTol tolerance to compute de root.

The following data is stored:

intIte iterations used to compute the root.

dblRoot approximation of the root.

dblError error in the approximation.

When two consecutive approximations are close enough, `dblTol`*, last approximation will be considered as good and will be stored in* `*biaRealRoot *ptRoot` *in* `dblRoot`*.*

Usage example

To approximate a root for the function $f(x) = \sqrt{2}$ to C functions need to be created. A C function for the mathematical function implementation:

```
/* f(x) = x^2 - 2 */
int myfunc(double x0, double *fx0) {
  *fx0 = (double)(x0 * x0 - 2.);
  return BIA_TRUE;
}
```

A C function for the mathematical derivative function implementation:

```
/* f'(x) = 2*x */
int myfuncder(double x0, double *fx0) {
  *fx0 = 2.*x0;
  return BIA_TRUE;
}
```

Both functions must meet the following requirements:

- An integer value is returned, **BIA_TRUE** when function is evaluated in $x0$ and **BIA_ZERO_DIV** if a division by zero takes place.
- $x0$ value to evaluate the function.
- $*fx0$ pointer to a double to store the function's value in $x0$.

So to approximate function's root using Newton Method:

```
i = newtonMethod(&myRoot, &myfunc, &myfuncder);
```

Chapter 10

Runge-Kutta methods (rngkutta.h)

10.1 Introduction

Runge-Kutta are a family of implicit and explicit iterative methods used to approximate solutions of ordinary differential equations or **ODE**.

Butcher matricial notation is used in this implementation.

10.2 Data structures

10.2.1 `biaButcherArray` data structure

This structure is used to store the Butcher matricial notation.

Data structure is defined in figure 10.1 where:

intStages method stages.

***dblC** c_i coefficients stored in an array with size `intStages`.

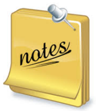
***dblB** b_i coefficients stored in an array with size `intStages`.

****dblMatrix** matrix to store $a_{i,j}$ method's coeficients.

```
typedef struct {
  double   *dblC,
           *dblB,
           **dblMatrix;

  int      intStages;
} biaButcherArray;
```

Figure 10.1: biaButcherArray data structure.



See appendix A if you need information about Butcher matricial notation.

10.2.2 `biaDataRK` data structure

This structure is used to store all the data needed to apply a Runge-Kutta method.

Data structure is defined in figure 10.2 where:

intNumApprox number of approximations to be done (size of the array `dblPoints`).

intImplicit when the Runge-Kutta method is implicit or not. The following constants are defined in the header file:

Name	Value
BIA_IMPLICIT_RK_TRUE	0
BIA_IMPLICIT_RK_FALSE	1

***dblPoints** array with dimension `intNumApprox` and its elements will be the approximations in x_i where:

$$x_i = dblFirst + i \cdot dblStepSize \qquad \text{where} \qquad 0 \leq i < intNumApprox$$

dblStepSize method's step-size.

dblFirst first point used to compute all the approximations. The value of the function in this point is known (initial condition).

dblLast last point in which approximations will be computed.

strCoefs variable of type biaButcherArray (section 10.2.1) storing Butcher matricial notation.

```
typedef struct {
  int intNumApprox,
      intImplicit;

  double  *dblPoints,
          dblStepSize,
          dblFirst,
          dblLast;

  biaButcherArray strCoefs;
} biaDataRK;
```

Figure 10.2: biaDataRK data structure.

10.3 Node number calculations

10.3.1 intNodeNumber function

This function gets the number of nodes that can be placed in an interval. All nodes are equidistant.

The definition of this function:

```
int intNodeNumber(double dblLong, double dblStepSize)
```

where:

dblLong interval length.

dblStepSize distance between two nodes.

The function returns the number of nodes that can be placed.

Arguments are supposed to be different from zero.

Arguments are supposed to be positive.

10.4 Explicit Runge-Kutta methods (scalar problems)

10.4.1 `ExplicitRungeKutta` function

This function solves an *I.V.P.* using an explicit Runge-Kutta method.

The definition of this function:

```
int explicitRungeKutta(biaDataRK *ptData,
                  double (*IVP)(double dblX, double dblY)
```

where:

ptData pointer to a **biaDataRK** variable[1]. This variable contains all the necessary data to solve the *I.V.P.*.

PVI C function's pointer to a function implementing the *O.D.E.*. This functions needs to have two double arguments and returns the value of the *I.V.P.*:

dblX point where we want to evaluate the *O.D.E.*

dblY *O.D.E.* value in `dblX` ($y_i \approx y(x_i)$).

The following codes are returned:

BIA_MEM_ALLOC	Memory error allocation
BIA_TRUE	Success

Usage example

For instance, to solve this *I.V.P.*:

$$\begin{cases} y' = y(x) * \frac{x-y(x)}{x^2} \\ y(1) = 2 \end{cases}$$

the implementation of the `IVP` would be:

```
double IVP(double dblX, double dblY) {
  double  dblResultado;

  dblRes = dblY*((dblX-dblY)/(dblX*dblX));

  return (dblRes);
}
```

[1] Section (10.2.2).

intResultado = ***ExplicitRungeKutta****(&varstrDatRK, PVI);*

Resolvería el *P.V.I.* representado por la función *PVI* utilizando los datos almacenados en la variable, del tipo *DatosRK*, *varstrDatRK* y almacenaría en *intResultado* el código devuelto por la función.

10.4.2 `RungeKuttaClasico` function

Función que inicializa los coeficientes para el método *Runge-Kutta Clásico*, el cual es un método de 4 etapas y orden 4.

La notación matricial del método es la siguiente:

$$\begin{array}{c|cccc} 0 & 0 & & & \\ \frac{1}{2} & \frac{1}{2} & 0 & & \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

El prototipo de esta función es el siguiente:

int ***RungeKuttaClasico****(DatosRK *ptstrDatos)*

ptstrDatos puntero a una variable de *estructura* del tipo *DatosRK*.

La función devuelve los siguientes códigos:

ERR_AMEM	Hubo un error en la asignación de memoria.
TRUE	Se inicializaron con éxito los coeficientes.

Por ejemplo:

intResultado = ***RungeKuttaClasico****(&varstrDatRK);*

Inicializaría los coeficientes del método en la variable *varstrDatRK*, en *intResultado* el valor **TRUE** si se pudieron inicializar los coeficientes y en caso contrario **ERR_AMEM**.

10.4.3 MetodoHeun

Función que inicializa los coeficientes para el método de *Heun*, el cual es un método *Runge-Kutta* de 3 etapas y orden 3.

La notación matricial del método es la siguiente:

$$\begin{array}{c|ccc} 0 & 0 & & \\ \frac{1}{3} & \frac{1}{3} & 0 & \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ \hline & \frac{1}{4} & 0 & \frac{3}{4} \end{array}$$

El prototipo de esta función es el siguiente:

int ***MetodoHeun****(DatosRK *ptstrDatos)*

ptstrDatos puntero a una variable de *estructura* del tipo *DatosRK*.

La función devuelve los siguientes códigos:

ERR_AMEM	Hubo un error en la asignación de memoria.
TRUE	Se inicializaron con éxito los coeficientes.

Por ejemplo:

intResultados = ***MetodoHeun****(&varstrDatRK);*

Inicializaría los coeficientes del método en la variable *varstrDatRK*, en *intResultado* el valor **TRUE** si se pudieron inicializar los coeficientes y en caso contrario **ERR_AMEM**.

10.4.4 MetodoKutta

Función que inicializa los coeficientes para el método de *Kutta*, el cual es un método *Runge-Kutta* de 3 etapas y orden 3.

La notación matricial del método es la siguiente:

$$\begin{array}{c|ccc} 0 & 0 & & \\ \frac{1}{2} & \frac{1}{2} & 0 & \\ 1 & -1 & 2 & 0 \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array}$$

El prototipo de esta función es el siguiente:

int ***MetodoKutta****(DatosRK *ptstrDatos)*

ptstrDatos puntero a una variable de *estructura* del tipo *DatosRK*.

La función devuelve los siguientes códigos:

ERR_AMEM	Hubo un error en la asignación de memoria.
TRUE	Se inicializaron con éxito los coeficientes.

Por ejemplo:

intResultado = ***MetodoKutta****(&varstrDatRK);*

Inicializaría los coeficientes del método en la variable *varstrDatRK*, en *intResultado* el valor **TRUE** si se pudieron inicializar los coeficientes y en caso contrario **ERR_AMEM**.

10.4.5 `EulerModificado` function

Función que inicializa los coeficientes para el método de *Euler modificado*, el cual es un método *Runge-Kutta* de 2 etapas y orden 2.

La notación matricial del método es la siguiente:

$$\begin{array}{c|cc} 0 & 0 & \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}$$

El prototipo de esta función es el siguiente:

int ***EulerModificado****(DatosRK *ptstrDatos)*

ptstrDatos puntero a una variable de *estructura* del tipo *DatosRK*.

La función devuelve los siguientes códigos:

ERR_AMEM	Hubo un error en la asignación de memoria.
TRUE	Se inicializaron con éxito los coeficientes.

Por ejemplo:

intResultado = ***EulerModificado****(&varstrDatRK);*

Inicializaría los coeficientes del método en la variable *varstrDatRK*, en *intResultado* el valor **TRUE** si se pudieron inicializar los coeficientes y en caso contrario **ERR_AMEM**.

10.4.6 `EulerMejorado` function

Función que inicializa los coeficientes para el método de *Euler mejorado*, el cual es un método *Runge-Kutta* de 2 etapas y orden 2.

La notación matricial del método es la siguiente:

$$
\begin{array}{c|cc}
0 & 0 & \\
1 & 1 & 0 \\
\hline
 & \frac{1}{2} & \frac{1}{2}
\end{array}
$$

El prototipo de esta función es el siguiente:

*int **EulerMejorado**(DatosRK *ptstrDatos)*

ptstrDatos puntero a una variable de *estructura* del tipo *DatosRK*.

La función devuelve los siguientes códigos:

ERR_AMEM	Hubo un error en la asignación de memoria.
TRUE	Se inicializaron con éxito los coeficientes.

Por ejemplo:

*intResultado = **EulerMejorado**(&varstrDatRK);*

Inicializaría los coeficientes del método en la variable *varstrDatRK*, en *intResultado* el valor **TRUE** si se pudieron inicializar los coeficientes y en caso contrario **ERR_AMEM**.

Todas estas funciones suponen que la variable de *estructura*, del tipo *DatosRK*[2], no tienen dimensionados los punteros en ella contenidos, razón por la cual será necesario liberar la memoria asignada a estos antes de pasarle como parametro una variable de este tipo a una de las siguientes funciones(siempre y cuando se hayan dimensionado dichos punteros).

Hay que destacar que **NO** se inicializan todos los miembros de esta estructura, sólo aquellos miembros que contienen los coeficientes del método.

Los siguientes miembros **NO** se inicializan:

intNumAprox

[2] Apartado (**??**) en la página **??**

dblPuntos

dblPaso

dblInicio

dblFinal

Estos miembros son independientes del método, dependen del problema que se quiera resolver y tendrán que ser inicializados por el usuario.

Appendix A

Runge-Kutta methods

This appendix is intended to help to know how Runge-Kutta methods are implemented and used in this library.

A.1 What is a Runge-Kutta method?

Runge-Kutta methods are a family of numerical methods to approach solutions of ordinary differential equations (O.D.E). These methods are iterative methods used to solve "*initial problem value*" (**I.P.V**) or "*Cauchy problem*".

These methods are only-one-step methods with a fixed size for the method step[1].

A.1.1 What is a I.V.P.?

An *I.V.P.* is:

$$\begin{cases} y' = f(x, y(x)) \\ y(x_0) = y_0 \end{cases} \tag{A.1}$$

So y' is a function depending on the variable x, and the function $y(x)$. $y(x)$ is the solution of the equation A.1 and the point (x_0, y_0) belongs to the curve $y(x)$.

Solving the *I.V.P.* A.1 is finding a function $y(x)$ such as the equation A.1 is met.

[1]It is also possible to implement methods with a variable step known as *embedding*.

An example of a *I.V.P.*:

$$\begin{cases} y' = \frac{x*y(x)-y(x)^2}{x^2} \\ y(1) = 2 \end{cases} \tag{A.2}$$

The solution of the A.2 will be:

$$y(x) = \frac{x}{\frac{1}{2} + \ln x} \tag{A.3}$$

A.2 Runge-Kutta's method notation

$y(x_i)$ will be the exact value of the function $y(x)$ evaluated in x_i.

y_i will be the approximation of the function $y(x)$ in the point x_i.

h is the step used by the method in each iteration.

A.2.1 General formulation

A s-stages **Runge-Kutta**'s method formulation is:

$$y_{n+1} = y_n + h \cdot \sum_{i=0}^{s-1} b_i \cdot k_i \tag{A.4}$$

where:

$$k_i = f(x_n + c_i \cdot h, y_n + h \cdot \sum_{j=0}^{s-1} a_{i,j} \cdot k_j) \tag{A.5}$$

satisfying:

$$\sum_{j=0}^{s-1} a_{i,j} = c_i \tag{A.6}$$

A.2.2 Matricial notation (Butcher's)

Matricial notation is used to represent method's coeficients using a matrix.

For a s-stages **Runge-Kutta** method the matricial notation will be:

$$\begin{array}{c|ccc} c_0 & a_{0,0} & \cdots\cdots & a_{0,s-1} \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ c_{s-1} & a_{s-1,0} & \cdots\cdots & a_{s-1,s-1} \\ \hline & b_0 & \cdots\cdots & b_{s-1} \end{array}$$

In section 10.2.1 is shown a data structure used to store the Butcher array.

A.3 Runge-Kutta types

There are several types of **Runge-Kutta** methods.

A.3.1 Implicit Runge-Kutta

A **Runge-Kutta** method is said to be implicit when the $a_{i,j} \neq 0$ for some $j > i$.

The 2-stages Gauss method is an implicit **Runge-Kutta** method of 2-stages:

$$\begin{array}{c|cc} \frac{3-\sqrt{3}}{6} & \frac{1}{4} & \frac{3-2*\sqrt{3}}{12} \\ \frac{3+\sqrt{3}}{6} & \frac{3+2*\sqrt{3}}{12} & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

A.3.2 Semi-implicit Runge-Kutta

A **Runge-Kutta** method is said to be semi-implicit when the $a_{i,j} = 0$ when $j > i$.

A 2-stages semi-implicit **Runge-Kutta** method:

$$\begin{array}{c|cc} \frac{3+\sqrt{3}}{6} & \frac{3+\sqrt{3}}{6} & 0 \\ \frac{3-\sqrt{3}}{6} & \frac{-\sqrt{3}}{3} & \frac{3+\sqrt{3}}{6} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

A.3.3 Explicit Runge-Kutta

A **Runge-Kutta** method is said to be explicit when the $a_{i,j} = 0$ when $j \geq i$.

A 4-stages explicit **Runge-Kutta** method also known as "**classic Runge-Kutta**":

$$
\begin{array}{c|cccc}
0 & 0 & & & \\
\frac{1}{2} & \frac{1}{2} & 0 & & \\
\frac{1}{2} & 0 & \frac{1}{2} & 0 & \\
1 & 0 & 0 & 1 & 0 \\
\hline
 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6}
\end{array}
$$